ITEA 2

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

Σ! 3674

WORKPACKAGE 2: METHOD ENGINEERING

D2.6 V2

MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

Project acronym: UsiXML

Project full title: User interface eXtensible Mark-up Language

ITEA label n° 08026

D2.6 – MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

DOCUMENT CONTROL

Deliverable N°*: D2.6 V2*
Due Date*: 01/2013*
Delivery Date*: 01/2013*

Short Description*:*
This deliverable shows how the UsiXML models defined in WP1 are derived and mapped between themselves by the way of transformation rules. These rules are compatible with MDE and aim to support the user interface development method. Different transformation paths are supported including a transformation path in which an application domain model expressed in UML is the basis for the generation of the associated interaction models. This approach enables interactive application developers to capitalise on the work of system designers, since many existing systems are based on a domain model. Design knowledge is consolidated in the set of transformation rules presented in this document.

Lead Partner*: UPV*
Contributors: *UPV, PRO*
Made available to*: [Confidential/Restricted/Public see FPP]*

Rev	Date	Author	Checked by	Internal Approval	Description
0.0	06/05/2011	Nathalie Aquino (UPV), Ignacio Panach (UPV)			Initial version: table of contents
0.1	09/15/2011	Nathalie Aquino (UPV), Ignacio Panach (UPV)			Changes in the structure of the document according to suggestions from UJF and ICI. Incorporation of contributions from UJF, ICI, and UPV.
0.2	10/12/2011	Nathalie Aquino (UPV), Ignacio Panach (UPV)			Incorporation of improvements based on the review made by Guillermo Rodríguez (BIL)
0.3	28/01/2013	Nathalie Aquino (UPV), Ignacio Panach (UPV), Marc Gil (PRO)			-Identification of transformation rules among several models of UsiXML -Incorporation of ATL rules from Prodevelop (PRO)

D2.6 – MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

0.9	mm/dd/yy	Contributions integrated from: **1st Internal Reviewer**: Name, Organisation **2nd Internal Reviewer**: Name, Organisation)	Name, organisation	**1st Internal Reviewer**: Name, organisation **2nd Internal Reviewer**: Name, organisation	Summary of modifications
1.0	mm/dd/yy	Final deliverable			

D2.6 – MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

CONTENTS

D2.6 – MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

D2.6 – MODEL TRANSFORMATION BASED DEVELOPMENT PROCESS

1. EXECUTIVE SUMMARY

This document describes how models of the UsiXML framework are derived and mapped between themselves by means of transformation rules. All this process is compatible with MDE so as to support the user interface development method. The UsiXML framework defines the user interface by means of models at different abstraction levels and contexts. Therefore, it is necessary to specify how we can transform source models into target models. Different transformation paths are possible giving support to forward- and backward-oriented transformation processes. The possible transformation paths provide flexibility for developers. In order to guide the analyst throughout the different alternatives, this document includes some guides. Finally, design knowledge is consolidated in the set of transformation rules presented in this document.

2. DOCUMENTS

2.1. MANDATORY

UsiXML FPP V4.0	Full project proposal

2.2. REFERENCE

3. INTRODUCTION

This document describes how models of the UsiXML framework are derived and mapped between themselves by the way of transformation rules in a way that is compatible with MDE so as to support the user interface development method. Next, we define some concepts we use throughout the deliverable:

- Process: refers to a series of fixed actions to achieve a result.
- Method: refers to a general set of ordered steps to achieve a result. A Method may encompass several processes.
- Methodology: refers to the study of methods.

The UsiXML framework defines the user interface by means of models at different abstraction levels and contexts. Therefore, it is necessary to specify how we can transform source models into target models. Different transformation paths are possible giving support to forward- and backward-oriented transformation processes. There are four types of transformation rules:

- Reification: the abstraction level of the source meta-model elements is higher (further from the final interface) than the abstraction level of the target meta-model elements. The source and target contexts of use are the same. Example: a transformation from Abstract Interaction Model to Concrete Interaction Model.

- Abstraction: the abstraction level of the source meta-model elements is lower than the abstraction level of the target meta-model elements. The source and target contexts of use are the same. Example: a transformation from Concrete Interaction Model to Abstract Interaction Model.
- Reflection: the abstraction level of the source meta-model elements is equal to the abstraction level of the target meta-model elements. The source and target contexts of use are the same. Example: a transformation between two versions of an Abstract Interaction Model.
- Translation: the abstraction level of the source meta-model elements is equal to the abstraction level of the target meta-model elements. The source and target contexts of use are different. A transformation from a context for a Web application to a context for a PDA application.

There are different alternatives to perform a transformation. For example, an input element defined in the Abstract Interaction Model can be transformed into a Checkbox or into a Radiobutton in the Concrete Interaction Model. These alternatives make UsiXML flexible enough to develop systems according to users' requirements. The deliverable aims to define the alternative paths for all the transformations. The document is structured in the following sections:

- **Section 4 - Global Vision for Transformations: Models, Methodologies, and Paths**: there are many models that compose the UsiXML framework and not all of them can be transformed into other ones. This section briefly presents these models, explains methodologies for transformation processes, and identifies different development paths. Development paths are composed of a sequence of pairs (source models, target models).
- **Section 5 - Definition of Transformations**: for each pair (source models, target models), this section presents the transformation rules that specify the transformation.
- **Section 6 - Applying Usability Criteria to Choose Transformation Rules**: we can define several transformation rules to transform the same primitive of a source model into different primitives of a target model. Moreover, these rules can be contradictory or exclusive among them. Therefore, we need some criteria to decide which transformation rule must be chosen when there is more than one possibility. The aim of this section is to propose usability criteria to choose transformation rules.

4. GLOBAL VISION FOR TRANSFORMATIONS: MODELS, METHODOLOGIES, AND PATHS

This section briefly presents the models that compose the UsiXML framework. Furthermore, different methodologies for transformation processes together with different possible transformation paths are presented.

The models currently defined in the UsiXML project are the following:

- **Domain Model**: describes the various entities manipulated by a user while interacting with the system.
- **Task Model**: describes the interaction among tasks as viewed by the end user perspective with the system.
- **Context Model**: describes the relevant attributes of the entities that are part of the system environment. It focuses on the representation of these entity attributes; the recognition and individualization of particular entities according to the state of the defined attributes; and the different physical space representation of these entities.
- **Abstract User Interface Model (AUI)**: is an expression of the user interface in terms of interaction spaces (or presentation units), independently of which interactors are available and even independently of the modality of interaction (graphical, vocal, haptic …).
- **Concrete User Interface Model (CUI)**: is an expression of the user interface in terms of "concrete interaction units", that depend on the type of platform and media available and has a number of attributes that define more concretely how it should be perceived by the user.

- **Transformation Model**: gathers a set of transformation rules that together describe how models of one or more source meta-models are transformed into models of one or more target meta-models. A transformation model can also be used to specify how to improve models or how to customize them for different contexts of use.
- **Workflow Model**: describes the flow of tasks and information that are passed from one worker to another, according to a set of procedural rules. Is decomposed into processes that are in turn refined into tasks. Tasks are related to produced/consumed resources and to job definitions of the organization.
- **Business Process Model (SPEM4UsiXML)**: proposes the elements that are necessary to define any UsiXML methodology.

The UsiXML models correspond to different abstraction levels. Figure 4-1 illustrates some of the UsiXML models located at different abstraction levels as well as different transformation paths. Reification allows us to define forward-oriented development processes; abstraction allows us to define backward-oriented development processes; reflection allows us to improve models; and translation allows us to address different contexts of use.

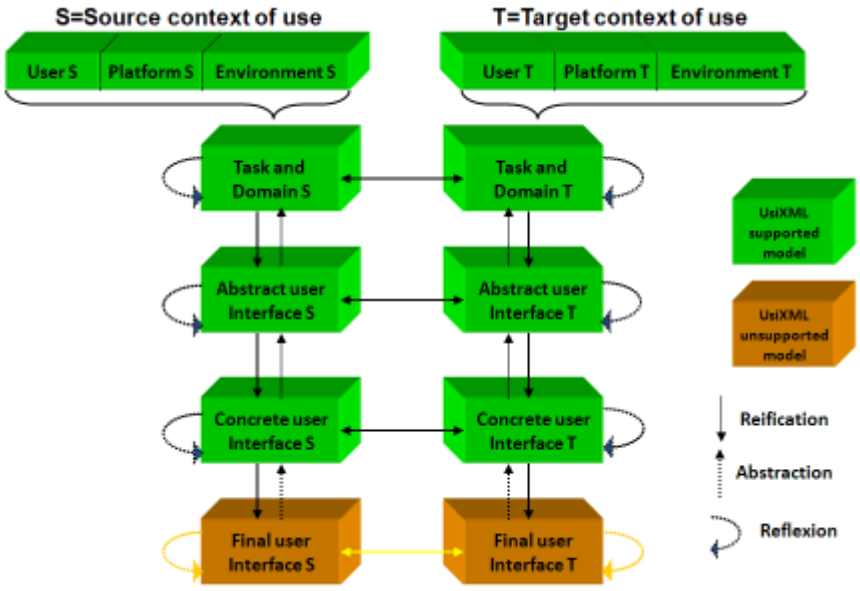

Figure 4-1. UsiXML models, abstraction layers, and development paths

Taking Figure 4-1 and other possibilities into account, some possible paths for UsiXML transformations are the following:

- Domain Model to Abstract User Interface Model (Domain → AUI)
- Abstract User Interface Model to Domain (AUI → Domain)
- Task Model to Abstract User Interface Model (Task → AUI)
- Abstract User Interface Model to Task Model (AUI → Task)
- Abstract User Interface Model to Concrete User Interface Model (AUI → CUI)
- Concrete User Interface Model to Abstract User Interface Model (CUI → AUI)
- Concrete User Interface Model to code (CUI → Code)
- Context Model to another Context Model (Context→Context)

Figure 4-1 shows all the possible transformations that can be performed in UsiXML. This diagram shows a linear process including several models and metamodels. According to some surveys like [Garzotto 2007], [Fitzgerald 1998] or [Barry

2001], designers expect that methods offer a variety of paths for the creation of a system so that it becomes possible to elicit a process that suits well the project specific context. The same surveys make it explicit that designers anticipate the arrival of methods that can understand and adapt to a user's habits and know-how rather than having to manually adapt to the methods.

To face the needs of adaptation and flexibility in the design, the method engineering domain has promoted the notion of method fragments [Ralyté 2001], [Brinkkemper 1998]. These fragments can be reused, selected and combined in order to answer specific project needs [Crescenzo 2009], [Perez-Medina 2010]. The main problems of this approach are on one hand the selection of appropriate fragments and on the other is the combination of these fragments. To facilitate the achievement of these important stages, some authors like [Rolland 2006] have proposed to elicit one strategy – and thereby one method fragment – within the set of strategies that can achieve a specific goal. However fragments combination remains an open question. To reduce this drawback, it is important to promote the flexibility for the designer. They propose a method based on a spinal column where flexibility will be added through the use of goals. Therefore, it will be possible to elicit a path where each step is responsible for one goal. Several activities can be proposed for each step if they answer to the same goal. For example, several HCI activities such as organizing focus groups or card sorting can answer the goal "study the user interaction". In this example, each card is a part of the spinal column.

This flexibility is at the heart of the work of Eric Céret [Cer11] where there is the desire to make it possible for a designer to

- use the knowledge the designer is familiar with (e.g. UML sequence diagram instead of UsiXML task model),
- partially model the system under study (e.g. replace the platform model by a provided default platform model; avoid a part of a task model and use a mock-up that will be transformed into a CUI directly),
- choose the most suitable strategy in a set of proposed ways to reach the same goal (e.g. create a domain model or automatically extract it from a database).

Within this scenario, we aim to promote flexibility for the designer and self-explanation for the end-user, both based on a graph of models at runtime [Cer11]. The vision is to make it possible for the designer to

- use the knowledge the designer is familiar with (e.g. UML sequence diagram instead of UsiXML task model),
- partially model the system under study (e.g. replace the platform model by a provided default platform model; avoid a part of a task model and use a mock-up that will be transformed into a CUI directly),
- choose the most suitable strategy in a set of proposed ways to reach the same goal (e.g. create a domain model or automatically extract it from a database).

Mappings between models and transformations should be guided by quality. A vision is to ensure the designer will make the design choices explicit. This design rationale is used at runtime so as to provide the end-user with explanations of the user interface (e.g., What is this button for? How can I enable it?). See the work [GCD+11] for further information.

5. DEFINITION OF TRANSFORMATIONS

For each of the possible paths identified in Section 4, this section elicits the transformation rules defined in UsiXML. We have grouped the transformation according to these four groups: Reification, Abstraction, Reflection and Translation.

5.1. Reification

Reification gathers all the transformations from a source model more abstract than the target model. Next, we present these transformations among all the UsiXML models. All these transformation rules are presented with classes of the metamodels explained in Deliverable 1.3.

5.1.1. Domain Model to Abstract Model (Domain → Abstract)

This section presents some rules to transform elements of the Domain Model into the Abstract User Interface Model.

1. DomainModel can be transformed into AbstractUIModel. Domain Model is the class that represents the domain. It can be transformed into the class that represents the Abstract Model.
2. Classifier can be transformed into AbstractCompoundIU. Classifier represents interfaces and classes in the domain model. Classifier can be converted into an AbstractCompoundIU in the Abstract Model, which represents also interfaces and a set of information.
3. Property can be transformed into AbstractDataIU. Property represents the properties of the classes and interfaces. This information can be transformed into the class AbstractDataIU, which was defined for this goal in the Abstract Model.
4. Operation can be transformed into Rule. Operation represents the methods that compose the class. These methods are represented in the Abstract model with the class Rule. Therefore, we can define a transformation between these two classes.
5. Relationship can be transformed into Rule. Relationship represents a relation between two classes of the domain. This relation can be seen as a navigation between two interaction units. These navigations are specified in the Abstract Model with rules. Therefore, from the class Relationship we can obtain some Rules.

5.1.2. Task Model to Abstract User Interface Model (Task → Abstract)

This section presents some rules to transform elements of the task Model into the Abstract User Interface Model:

1. TaskModel can be transformed into AbstractUIModel. TaskModel is the class that represents the whole Task Model, therefore, it can be transformed into the class that represents the whole Abstract Model (AbstractUIModel).
2. Task can be transformed into AbstractInteractionUnit, AbstractListener, Rule and Condition. The class Task in the Task Model is the main entity of the model. This class represents a task which can be triggered by the end-user. We can consider that each task defined in the Task Model corresponds to an action in the Abstract Model. In the Abstract Model, actions are represented inside Abstract Interaction Units, which has an Abstract Listener which is waiting for actions from the user. The attribute Postcondition of the class Task can be represented in the Abstract Model with the class Rule. The class Precondition of the class Task can be transformed into the class Condition, since this class represents the condition that must be fulfilled to execute the rule related to the Abstract Listener.

5.1.3. Abstract User Interface Model to Concrete User Interface Model (Abstract → Concrete)

Next, we present some rules to transform elements of the Abstract User Interface Model into the Concrete User Interface Model.

1. AbstractUIModel can be transformed into ConcreteUIModel. AbstractUIModel is the class that represents the whole Abstract Model, therefore, it can be transformed into the class that represents the whole Concrete Model (ConcreteUIModel).
2. AbstractInteractionUnit can be transformed into ConcreteInteractionUnit. AbstractInteractionUnit represents any abstract object, while ConcreteInteractionUnit represents any concrete object. Therefore, we can perform this transformation. Apart from the transformation, we can also specialize the object ConcreteInteractionUnit in: ConcreteMultimodalIU, ConcreteGraphicalIU, ConcreteVocalIU, ConcreteTactileIU, depending on how the user is going to interact with the system.
3. AbstractLocalization can be transformed into ConcreteLocalization. AbstractLocalization is used to represent text attributes for AbstractInteractionUnits. This class can be transformed into ConcreteLocalization which aims to define the language of specific interaction units.
4. AbstractCompoundIU can be transformed into ConcreteGraphicalCompoundIU. AbstractCompoundIU is used to gather several AbstractInteractionUnits. Its equivalent in the Concrete Model is ConcreteGraphicalCompoundIU.
5. AbstractCompoundIU can be transformed into Menu. If the AbstractInteractionUnit that represents the AbstractCompoundIU aims to be transformed into an organizer of menu items in the Concrete Model, we must transform the AbstractCompoundIU into Menu.
6. AbstractCompoundIU can be transformed into MenuBar. If the AbstractInteractionUnit that represents the AbstractCompoundIU aims to be transformed into an organizer of menu bar items in the Concrete Model, we must transform the AbstractCompoundIU into MenuBar.
7. AbstractCompoundIU can be transformed into MenuBarItem. If the AbstractInteractionUnit that represents the AbstractCompoundIU aims to be transformed into a menu item in the Concrete Model, we must transform the AbstractCompoundIU into MenuItem.
8. AbstractCompoundIU can be transformed into MenuItem. If the AbstractInteractionUnit that represents the AbstractCompoundIU aims to be transformed into a menu bar item in the Concrete Model, we must transform the AbstractCompoundIU into MenuBarItem.
9. AbstractTriggerIU can be transformed into CommandButton. AbstractTriggerIU is used to represent that an action can be triggered. If the action must be triggered with a button, the class AbstractTriggerIU must be transformed into CommandButton in the Concrete Model.
10. AbstractTriggerIU can be transformed into MenuBarItem. If the action to trigger is displayed in a menu bar, the class AbstractTriggerIU is transformed into the class MenuBarItem.
11. AbstractTriggerIU can be transformed into MenuItem. If the action to trigger is displayed in a menu, the class AbstractTriggerIU is transformed into the class MenuItem.
12. AbstractDataIU can be transformed into ConcreteGraphicalElementaryIU. AbstractDataIU is used to link the Domain Model with the Abstract Model. The relationship between the Domain Model and the Concrete Model is performed by means of ConcreteGraphicalElementaryIU. Therefore, we can transform AbstractDataIU into ConcreteGraphicalElementaryIU. In the transformation, this ConcreteGraphicalElementaryIU can be specialized in: ImageComponent, Slider, FilePicker, ProgressionBar, VideoComponent, AudioComponent, Toolbar, CommandButton, and Label. The specialization depends on the functionality that represents the AbstractDataIU.
13. Rule can be transformed into ConcreteGraphicalListener. Rule is a class that represents an abstract listener. Therefore, from this abstract listener we can obtain a concrete one to represent an action or an event.
14. Rule can be transformed into ConcreteVocalListener, when the listener is vocal.
15. Rule can be transformed into ConcreteTactileListener, when the listener is tactile.

16. Justification can be transformed into ConcreteJustification. Justification is used in the Abstract Model as a motivation for the ECA rule. In the Concrete Model, the class ConcreteJustification also represents the motivation of the ECA rule but with a dependence on the interface where the ECA rule will be triggered.
17. Condition can be transformed into ConcreteCondition. Condition is used in the Abstract Model as a condition that must be satisfied to trigger an action. ConcreteCondition has the same goal but with a dependence on the interface where the action will be triggered.

5.1.4. Concrete User Interface Model to Code (Concrete → Code)

Next, we present some rules to transform elements of the Concrete Model into code (model to code transformations).

1. ConcreteGraphicalIU can be transformed into an interface. This transformation can be performed only if the ConcreteGraphicalIU is not a component of other instance of ConcreteGraphicalIU.
2. ConcreteGraphicalIUs which are specialized in instances of the class ConcreteGraphicalElementaryIU, can be transformed into visual elements inside the interface. The possible target elements can be: an image, a video, an audio, a slider, a file picker, a progression bar, a tool bar, a command button, a label, a custom widget, a tab bar, a menu bar, a menu, a combobox, a checkbox, a radiobox or a listbox. The specific visual element in which the ConcreteGraphicalElementaryIU is transformed depends on the specialization of this class in the Concrete User Interface Model.
3. ToolBarItem is transformed into an item of the ToolBar.
4. ToolBarSeparator is transformed into a separator of the ToolBar.
5. Tab is transformed into a tab inside a bar of several tabs.
6. MenuBarItem is transformed into an item of a MenuBar.
7. MenuItem is transformed into an item of the Menu.
8. MenuSeparator is transformed into a graphical separator between two items of the menu.
9. ComboItem is transformed into an item of a ComboBox.
10. CheckItem is transformed into an item of a CheckBox.
11. RadioItem is transformed into an item of a RadioBox.
12. LisItem is transformed into an item of a ListBox.
13. TableLayout, Row and Cell can be transformed into a graphical table. Each instance of the class Row is a row of the table, and each instance of the class Cell is a cell of the row.
14. The class ConcreteGraphicalStyle can be used to define the colours and fonts used throughout the whole system.
15. The class ConcreteGraphicalListener and all its components, ConcreteJustification, ConcreteCondition, ConcreteEvent and ConcreteAction can be transformed into an action which implements an ECA rule.

5.2. Abstraction

This type of transformation gathers rules to transform from a source model less abstract than the target model. These transformations are the same that the transformations of Reification but applied inversely. This is the reason why we do not define these transformations explicitly.

5.3. Reflection

This type of transformations represents changes in the same model. This is useful to store different versions of a model. These transformations can be applied to all the models that compose UsiXML, and we should store every change in any model. We do not specify these transformations because they have the same structure for all the classes. The source is the class which has changed between two versions and the target is the same class in the new version.

5.4. Translation

This type of transformation is used to adapt the system to a different context from which it was defined. In this case, the source and the target model is the Context Model. Next, we present the possible transformations between two versions of the Context Model:

1. Property: this class represents an attribute of a capability, such as the VerticalResolution or the HorizontalResolution of a DisplayCapability. Properties can change between two contexts. Moreover, if a property change, we must update the class PropertyConstraint, which defines the constraints of the property.
2. Entity: this class represents the situation where the system operates. There are two types: Human and Non-Human. We can swap these values when we change the context. A change in Entity also affects all the instances of EntityExtension related to it.
3. Capability: this class represents a characteristic that can be shared among several entities. This is usually used to represent aspect descriptors (such as WiredNetworkCapability or DisplayCapability). These descriptors can change between two contexts.
4. EntityState: this class represents a state of an entity with regard to the environment that is relevant to the system operation. The state of the entity can be modified between two contexts. Changes in this class can also affect the class StateExtension, since this class represents the relationship between two EntityStates.
5. Observation: this class relates Entity and EntityState. If at least one of these classes changes between two contexts, the class Observation must be updated according to these changes.
6. PropertyConstraint: this class represents the constraints related to the properties. These constraints can change between two contexts.
7. Zone: this class represents physical and virtual spaces in the environment, which can change between two contexts. A change in Zone also involves changes in the class Representation, which relates EntityState to Zone.
8. Situation: this class represents a space-time relationship among the different EntityStates of the entities that are part of the Entities that affect the system operation. This relationship can change between two contexts.
9. Expression: this class represents relationships among EntityStates in space and time dimensions. Both dimensions can change between two contexts.

5.5. The ICI Approach

Previous subsections explains transformations between two models which are in adjacent abstraction layers (for instance, from the Abstract User Interface Model to the Concrete User Interface Model). However, the UsiXML framework allows working with transformations between non-adjacent models. As example, this section presents the transformation-based approach followed by the Institutul National de Cercetare-Dezvoltare in Informatica – ICI Bucuresti (ICI) for the development of interactive systems. The Concrete User Interface Model is produced by transformations from task and

domain models (avoiding the Abstract User Interface Model). The transformation rules used in the approach are presented and exemplified.

5.5.1. General approach and methodology

In this approach, the concrete model is produced by transformations from task and domain models. This is consistent with the flexibility principle of UsiXML methodology that makes it possible to accommodate different development paths. Therefore, it is possible to skip the abstract level which is not deemed necessary for the current context. The rationale forthe ICI approach is explained below.

According to Norman [Nor91] changing the artefact is changing the nature of the task. In a task-based approach to user interface development, the task model is given a leading role among other models and the task design is a key activity in the design of a usable user interface. In a previous work [Pri06b] three decomposition layers were identified, which are relevant in task modelling for user interface design:

- A functional layer may be created when considering application functions with respect to user tasks.
- A planning layer results from the decomposition of functional tasks that are independent to the constraints imposed by a target hardware and software platform.
- An operational layer results from the decomposition of unit tasks, which are, carried on by using interaction techniques available on a target platform.

In [Pri06b] we have also identified several task-domain mappings that are useful for the model-based derivation of the concrete user interface:

- Unit tasks – domain objects: operations performed on domain objects are modelled as unit tasks in the task model.
- Basic tasks – domain object attributes / available commands: operations performed on domain object attributes are mapped onto information control basic tasks while available commands on the target platform are mapped onto function control basic tasks.

To ensure usability, task modelling should go up to the level of basic tasks.

The transformation rules used in this approach are producing task patterns that could be further exploited along for similar applications. Therefore, we consider that transformation rules should be embodied in a set of tools that are producing UsiXML specifications. In order to make the specification manageable with a reasonable effort, several functions are required for such a tool: opening a model, finding a model element, inserting the specification of a model element into the model, finding a specification element (tag / model element attribute), and editing a specification element.

These functions should be agreed in the consortium since they are adding an overhead that is part of the transformation rules. In the following sub-sections we will describe the transformation rules that create / modify model element, with a minimal overhead that refers to these functions. This overhead includes conditions and repetitions.

In UsiXML the task temporalization should be specified as a relationship between two tasks. However, at this stage is difficult since is requiring an overhead to specify the tasks. Therefore, we specified the temporal operator as a task attribute like in CTTE. In this respect, the last task on a decomposition level does not have a temporal operator.

5.5.2. Example

In order to illustrate the transformation rules we will use an application previously developed in a task based approach [Pri09]. The main goal of the application is the management of usability problems and it provides a software assistant for formative usability evaluation. The purpose of formative usability evaluation is to identify and fix usability problems as early as possible in the development life cycle. There are two methods used: inspection-based evaluation (heuristic

evaluation, guideline-based evaluation) and user testing. In each case, a heuristic or design guideline is used to document the usability problem.

Usability problems are recorded for each task (aggregation relationship). Usability problems are the central class in this application. Heuristics are used to document a usability problem (association relationship). Guidelines are used to detail a heuristic (specialization relationship). The heuristics are displayed in a list box. When a heuristic is selected, its definition is displayed bellow the list box. Several heuristics could be associated to a usability problem. The user has the ability to consult the guidelines detailing a heuristic in a separate window (by pressing the ShowGuidelines button).

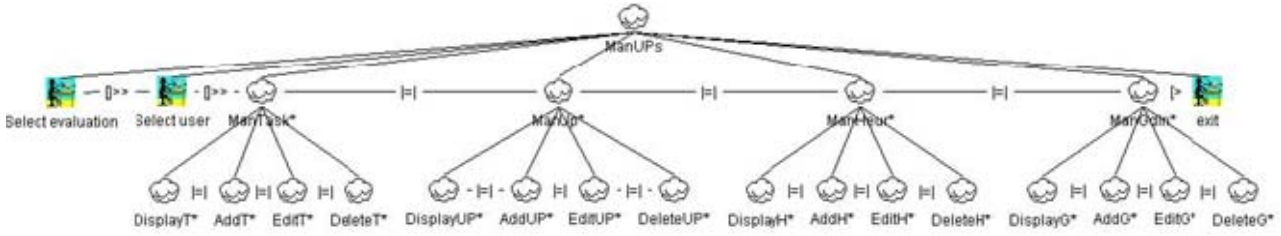

Figure 5-1. The functional task layer

In Figure 5-1 the functional task layer is presented using the CTT notation [Pat99].

The functional layer shows the business goals that should be accomplished with the target application. Each function is mapped onto a user task. In this example, the tasks are accomplishing the management of four classes: tasks, usability problems, heuristics and guidelines.

In Figure 5-2 the interaction unit for editing a usability problem is presented. This corresponds to the functional task 'EditUP' in Figure 5-1.

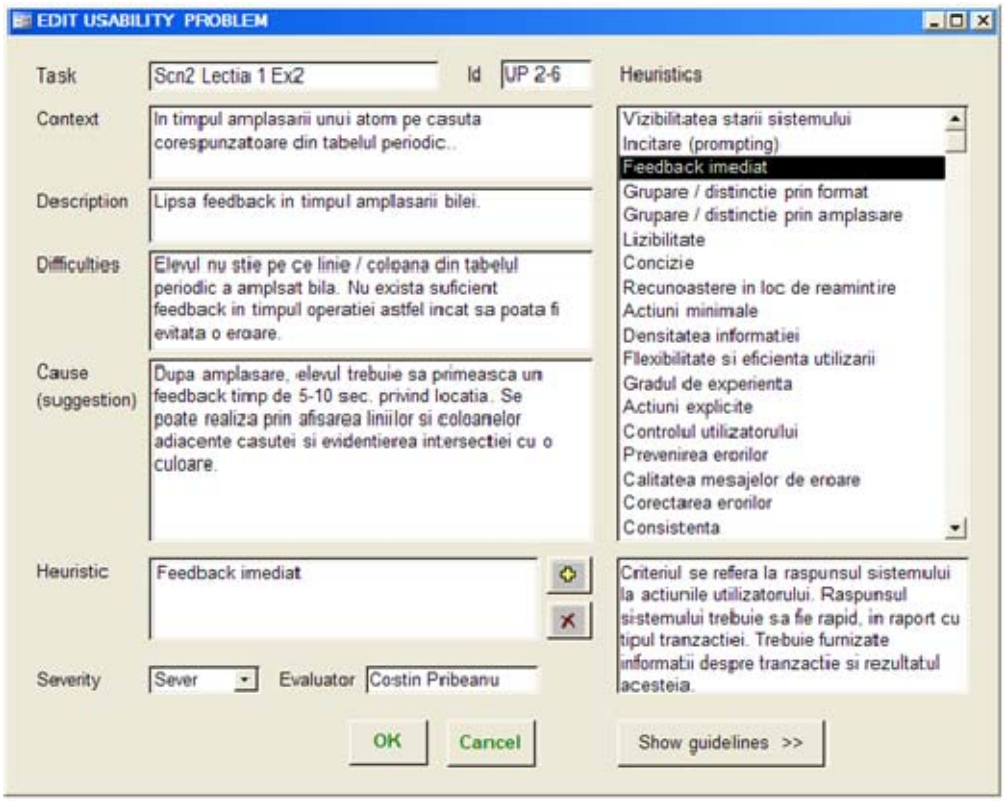

Figure 5-2. Editing a usability problem

Ideally, the user interface illustrated in Figure 5-2 should be created by applying transformation rules to the task and domain models. In a previous work [Pri06b] mapping rules were proposed to produce the operational task layer. In this work it is intended to define transformation rules that cover the whole hierarchy in the task model.

5.5.3. Transformations from Task Model to Task Model (Task → Task)

In this transformation the source is the task model and the destination is also the task model (reflection). According to our methodology, there are three layers in the task model: functional, planning and operational. The task to task transformation is used to produce the first level of decomposition in the task model based on task requirements.

The source is a task having as goal the management of a class (e.g. usability problems in Figure 5-3). The transformation is producing abstract tasks that correspond to operations performed onto the domain objects. Nevertheless, the domain model itself is not used since these operations are generic (such as display, create, edit or delete) and the decision to include them in the task model depends on the user's goals. In this respect, the domain model is only informing the task model as regarding the classes that are used in the application.

We identified four transformations that produce the first level of task decomposition in the task model for a given class, one for each operation type. The effectiveness of automating this transformation depends on the complexity of the target application given the fact that it is relatively easy to produce the destination interactively with a task editor (like CTTE).

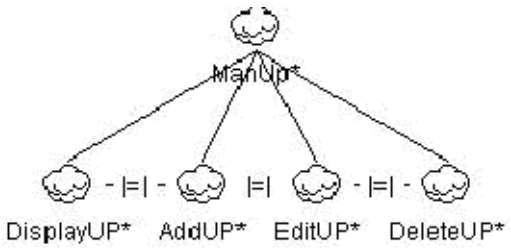

Figure 5-3. Task to task transformation at functional layer

For each class in the domain model a transformation is needed. In our example, we have four classes: tasks, usability problems, heuristics, and guidelines.

In order to produce the task model illustrated in Figure 5-1 we could apply a second transformation that integrates the results of the four previous transformations. An alternative is to specify the first level manually with a task editor and then to apply the transformation rule for each functional task.

The transformations rules are using two attributes of the task in the source task model: the canonical task type and the nature of the task.

The canonical task type could be: Manage, Create, Edit, Delete, Display, Select, EditAttr, Consult, EditAssoc, AddAssoc, DelAssoc, Search etc. This attribute reflects operations upon objects, attributes, and relationships between objects. The nature of the task could be abstract (complex), function control (selecting an item or a command), information control (Edit or Display), user (a cognitive task or a task manipulating and external object).

The functional layer in the task model is application specific. In our example, the purpose is to manage several classes of objects. Each functional task on the first level of decomposition has the canonicalTaskType attribute set to 'Manage'.

The transformation function INSERT TASK is inserting a task specification after the current pointer in the model and assumes the availability of the parent task attributes. Since the transformation rules are applied for each class of objects, the corresponding information from the domain model is also available (class name, relationships, object attributes).

The next transformation rule is producing the first level of task decomposition. The rule is performed for each class in the domain model and takes as input the parent task (which is decomposed) and the class name.

```
RT1.1 IF canonicalTaskType = 'Manage' THEN
INSERT TASK, name = 'Display' + Class.name, canonicalTaskType = 'Display',
child-of = parentTask.id, tOperator = '|=|', iterative = [0…*], nature =
'Abstract'
INSERT TASK, name = 'Add'+ parentTask.name, child-of = parentTask.id,
canonicalTaskType = 'Create', tOperator = '|=|', iterative = [0…*],
nature='Abstract'
INSERT TASK, name = 'Edit'+ parentTask.name, child-of = parentTask.id,
canonicalTaskType = 'Edit', tOperator = '|=|', iterative = [0…*],
nature='Abstract'
INSERT TASK, name = 'Delete'+ parentTask.name, child-of = parentTask.id,
canonicalTaskType = 'Delete', iterative = [0…*], nature = 'Abstract'
ENDIF
```

The four tasks are iterative (infinite) and linked with t '|=|' temporal operator (could be performed in any order), like their parent tasks. These tasks are showing the last level of decompostion in the functional layer.

5.5.4. Transformations from Task Model and Domain Model to Task Model (Task + Domain → Task)

These transformations are producing the planning and operational layers in the task model. The purpose of the transformation is to advance the task decomposition up to the level of basic tasks.

The transformations producing the planning layer are exploiting the semantic relationships between objects. In our example, editing a usability problem is a complex task with several sub-goals: editing the attributes of the usability problem, displaying info about a heuristic and displaying / editing the heuristics associated with a usability problem. The task "Edit UP" has a hierarchical structure and is further decomposed in several sub-tasks corresponding to the aforementioned sub-goals. The task "Edit Task" has a simple structure since there are no relationships with other objects.

The first transformation is producing the next decomposition level in the task model. According to the goal hierarchy, we might have several levels in the decomposition of a unit task.

The transformation producing the operational layer is exploiting the object attributes. This is the last decomposition level in the task model.

When the task model is complete, a model check should be performed. For example, in CTTE there are two useful functions: model checking and model simulation. The first checks the validity of the task model as regarding the temporal operators. The second enables the simulation of task execution to see if the implemented task model is behaving as intended.

5.5.4.1. Planning layer in the task model

The second layer in the task model specifies the way a user plans the task execution by decomposing it in sub-tasks and assigning a temporal order for each. This layer does not depends on the concrete operators on the platform but depends on the general conception of the interface (the structure of compound interaction units) and on the work context (division of labour, specific rules and procedures). The goals on the last level of decomposition are related to unit tasks, having a clear significance for the user.

In our example we will take the case of the functional task having as goal to add a usability problem. This is a complex tasks that could be further decomposed in sub-tasks:

- Editing the attributes of a usability problem.
- Displaying the associated heuristics that document the usability problem.
- Editing the associations (associate a heuristic, delete an association)
- Consulting the available heuristics.
- Consulting the guidelines associated with a heuristic.

There are two relationships that are displayed for this task: UP-H (usability problem – heuristics) and H-Gdln (heuristics – guidelines). The first is permanently displayed and is editable (the user could add or delete an association) while the second is optional and read only (the user can show / hide guidelines associated with a given heuristic).

The transformation rule bellow produces the next decomposition level.

```
RT2.1 IF canonicalTaskType = 'Create' AND nature='Abstract' and parent-of =
none
INSERT TASK, name = 'SelAdd', canonicalTaskType = 'Create', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelCom'
INSERT TASK, name = 'DispUPH', canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '|=|', iterative = [0], nature = 'Abstract'
INSERT TASK, name = 'EditUPAttr', canonicalTaskType = 'EditAttr', child-of =
parentTask.id, tOperator = '|=|', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'AssocH', canonicalTaskType = 'AddAssoc', child-of =
parentTask.id, tOperator = '|=|', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'DelAssocH', canonicalTaskType = 'DelAssoc', child-of =
parentTask.id, tOperator = '|=|', iterative = True, nature = 'Abstract'
INSERT TASK, name = 'ConsultGdln', canonicalTaskType = 'ConsultGdln', child-
of = parentTask.id, tOperator = '[>', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'ok', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = '[]', nature = 'SelCom'
INSERT TASK, name = 'cancel', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = none, nature = 'SelCom'
ENDIF.
```

The result of applying this rule is presented in Figure 5-4. The model highlights the five sub-tasks.

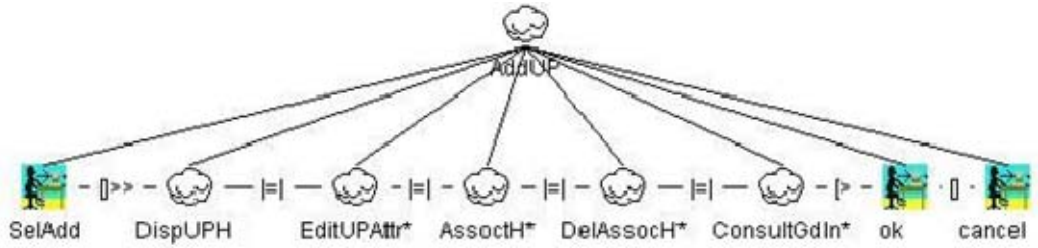

Figure 5-4. Task AddUP – planning level

This task has a similar structure with the task 'EditUP'. The next transformation rule produces the first decomposition level for the task ,EditUP'.

```
RT2.2 IF canonicalTaskType = 'Edit' AND nature = 'Abstract' and parent-of =
none THEN
INSERT TASK, name = 'SelUP', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelItem'
```

```
INSERT TASK, name = 'SelEdit', canonicalTaskType = 'Edit', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelCom'
INSERT TASK, name = 'DispUPH', canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '|=|', iterative = [0], nature = 'Abstract'
INSERT TASK, name = 'EditUPAttr', canonicalTaskType = 'EditAttr', child-of =
parentTask.id, tOperator = '|=|', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'AssocH', canonicalTaskType = 'AddAssoc', child-of =
parentTask.id, tOperator = '|=|', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'DelAssocH', canonicalTaskType = 'DelAssoc', child-of =
parentTask.id, tOperator = '|=|', iterative = True, nature = 'Abstract'
INSERT TASK, name = 'ConsultGdln', canonicalTaskType = 'ConsultGdln', child-
of = parentTask.id, tOperator = '[>', iterative = [0…*], nature = 'Abstract'
INSERT TASK, name = 'ok', canonicalTaskType = 'none', tOperator = '[]',
nature = 'SelCom'
INSERT TASK, name = 'cancel', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = none, nature = 'SelCom'
ENDIF
```

The difference between the two rules is the first sub-task to select the usability problem. Another way to produce the specification of this could be to modify the previous rule by inserting the first sub-task and changing the name of the second sub-task.

If no relationship is involved, a simpler task pattern is needed for editing a domain object. The next transformation rule is producing such a decompostion for editing the object 'Task'

```
RT2.3 IF canonicalTaskType = 'Edit' AND nature = 'Abstract' and parent-of =
none THEN
INSERT TASK, name = 'SelT', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelItem'
INSERT TASK, name = 'SelET', canonicalTaskType = 'Edit', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelCom'
INSERT TASK, name = 'EditTaskAttr', canonicalTaskType = 'EditAttr', child-of
= parentTask.id, tOperator = '[>', iterative = [0], nature = 'Abstract'
INSERT TASK, name = 'ok', canonicalTaskType = 'none', tOperator = '[]',
nature = 'SelCom'
INSERT TASK, name = 'cancel', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = none, nature = 'SelCom'
ENDIF
```

The result of applying this rule is illustrated in Figure 5-5. This task applies also for the class 'Evaluations'.

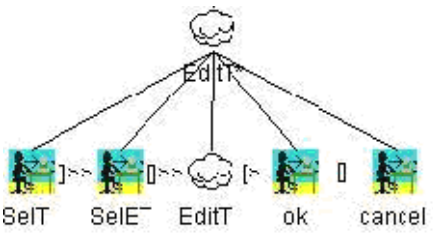

Figure 5-5. Task EditT – planning level

The next transformation rule is producing the next decomposition level for the task DelUP (delete a usability problem).

```
RT2.4 IF canonicalTaskType = 'Delete' and nature = 'Abstract' and parent-of
= none THEN
INSERT TASK, name = 'SelUP', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelItem'
INSERT TASK, name = 'SelDel', canonicalTaskType = 'Delete', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelCom'
INSERT TASK, name = 'DispUP', canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '[>', iterative = [0], nature = 'Abstract'
INSERT TASK, name = 'ok', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = '[]', nature = 'SelCom'
INSERT TASK, name = 'cancel', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = none, nature = 'SelCom'
ENDIF
```

The result of applying this rule is presented in Figure 5-6. The task DispUPH (see Rule 2.2) displays the onject attributes, the associated heuristics, and the list of available heuristics while the task DispUP displays only the attributes and associated heuristics. This task pattern applyes also for the other classes: tasks, evaluations and guidelines.

The next transformation rule is producing the first level of decomposition for the operation of displaying a usability

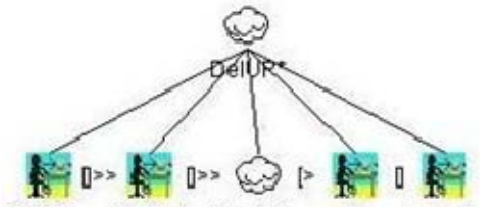

Figure 5-6. Task DelUP – planning level

problem.

```
RT2.5 IF canonicalTaskType = 'Display' and nature = 'Abstract' and parent-of
= none THEN
INSERT TASK, name = 'SelUP', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelItem',
DomainElement = UP-Class
INSERT TASK, name = 'SelDisp', canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '[]>>', iterative = [0], nature = 'SelCom'
INSERT TASK, name = 'DispUP', canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '[>', iterative = [0], nature = 'Abstract'
INSERT TASK, name = 'ok', canonicalTaskType = 'none', child-of =
parentTask.id, tOperator = 'none', nature = 'SelCom'
ENDIF
```

The result of applying this rule is presented in Figure 5-7. This task pattern applies also for the other classes: tasks, evaluations and guidelines. The task patterns created with the rules RT2.1 and RT2.2 apply for the specific relationships in the domain model.

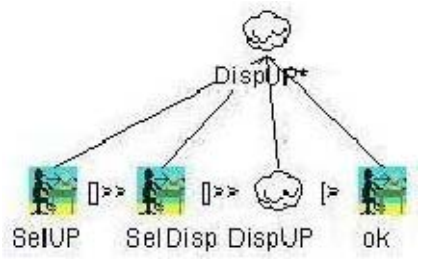

Figure 5-7. Task DispUP – planning level

5.5.4.2. Operational layer in the task model

The abstract tasks on the planning layer are unit tasks that are further decomposed in basic tasks. The source models are the domain and task model. The next rule produces the operational structure of the task DispUPH.

```
RT3.1 IF canonicalTaskType = 'DisplayAttr' and nature = 'Abstract' and
parent-of = none THEN
FOR EACH attribute in Class
IF IsVisible = True THEN
INSERT TASK, name = NameAttr, canonicalTaskType = 'Display', child-of =
parentTask.id, tOperator = '>>', nature = 'Display'
ENDIF
ENDFOR
TOperator = none
INSERT TASK, name = 'Assoc'+NameAssocClass, canonicalTaskType = 'Display',
child-of = parentTask.id, tOperator = '>>', nature = 'Display'
ENDIF
```

The temporal operator for the last task is 'none'. The result of applying this rule is presented in Figure 5-8.

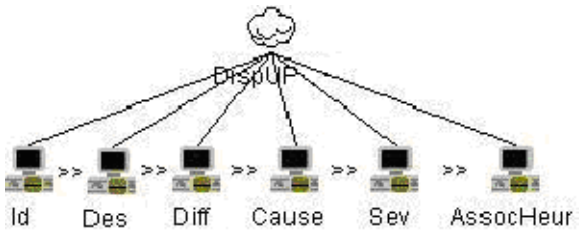

Figure 5-8. Task DispUP – operational level

```
RT3.2 IF canonicalTaskType = 'EditAttr' AND nature = 'Abstract' and parent-
of = none THEN
INSERT TASK, name = 'Disp'+NameClasaAssoc, canonicalTaskType = 'Display',
child-of = parentTask.id, tOperator = '>>', nature= 'Display'
INSERT TASK, name = 'Disp' + NameAssocClass + 'Des', canonicalTaskType =
'Display', child-of = parentTask.id, tOperator = '>>', nature = 'Display'
FOR EACH attribute in Class
IF IsVisible = True THEN
INSERT TASK, name = NameAttr, canonicalTaskType = 'Edit', child-of =
parentTask.id, tOperator = '>>', nature = 'Edit'
ENFOR
```

```
TOperator = none
ENDIF
```

The Operator for the last task is none. The result of applying this rule is presented in Figure 5-9.

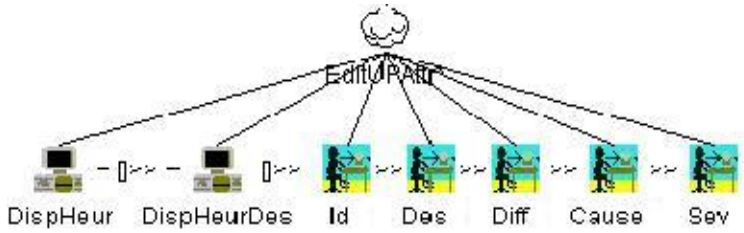

Figure 5-9. Task EditUP – operational level

The next rule is decomposing the task AddAssoc. After selecting a heuristic from the list, the definition is displayed and the user can associate it to the usability problem (optional task).

```
RT3.3 IF canonicalTaskType = 'AddAssoc' AND nature='Abstract' and parent-of
= none THEN
INSERT TASK, name = 'SelH', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', nature='SelItem'
INSERT TASK, name = 'DispHDes', child-of = parentTask.id, tOperator = '=>>',
nature='Display'
INSERT TASK, name = 'SelAssoc', child-of = parentTask.id, tOperator = none,
nature='SelCom', optional = true
ENDIF
```

The result of applying this rule is presented in Figure 5-10.

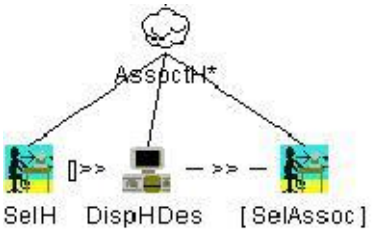

Figure 5-10. Task AssocH – operational level

```
RT3.4 IF canonicalTaskType = 'DelAssoc' AND nature = 'Abstract' and parent-
of = none THEN
INSERT TASK, name = 'SelHAssoc', canonicalTaskType = 'Select', child-of =
parentTask.id, tOperator = '[]>>', nature = 'f-Control'
INSERT TASK, name = 'DispHDes', child-of = parentTask.id, tOperator = '=>>',
nature='Display'
INSERT TASK, name = 'SelDelAssoc', child-of = parentTask.id, tOperator =
none, nature='SelCom', optional = true
ENDIF
```

The result of applying this rule is presented in Figure 5-11.

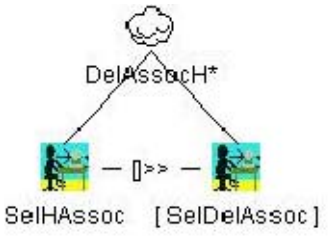

Figure 5-11. Task DelAssocH – operational level

Note that in order to associate a heuristic from the list of available heuristics is selected while for deletion, an associated heuristic is selected.

5.5.5. Transformations from Task Model and Domain Model to Concrete User Interface Model (Task + Domain → CUI)

In this case we have two sources (Task and Domain) and only one target (Task). These transformations are producing the concrete user interface.

In this approach, the operational layer in the task model is mapped on the concrete user interface.

5.6. Some Examples of Transformations in ATL

In previous subsections, we have defined transformation rules textually such a way they can be used in any transformation language. As illustrative example, we show in this subsection how some of these rules can be represented in a specific language. From all the existing languages, we use ATL [ATL], which is a language to specify transformations using a source metamodel and a target metamodel. The main reason why we have chosen ATL is because we already have the UsiXML metamodels in ecore (they were defined in D1.3). Next, we present some chunks of code to transform: Abstract User Interface Model into Concrete User Interface Model, Domain Model into Abstract User Interface Model, Concrete User Interface Model into code.

5.6.1. Transformation from AUI to CUI

We show some rules to transform AUI to CUI as example. The first rule transforms the class AbstractUIModel into the class ConcreteUIModel.

```
rule AbstractModel2ConcreteModel {
	from
		am : AUI!AbstractUIModel
	using {
		domainElement : OclAny = am.getElement(
			if AUI!AbstractDataIU.allInstances()->exists(ad |
ad.isElementaryIUTransformable()
				and ad.domainReference <> OclUndefined and ad.domainReference <>
'') then
```

```
                        AUI!AbstractDataIU.allInstances()->any(ad |
ad.isElementaryIUTransformable()
                                and ad.domainReference <> OclUndefined and
ad.domainReference <> '').domainReference
                    else
                        ''
                    endif
                    );
        }
        to
                cm : CUI!ConcreteUIModel (
                    interactionUnits <- am.compoundIUs
                )
        do {
                thisModule.model <- am;

                if (domainElement <> OclUndefined) {

        thisModule.createEAnnotation('usixml.cui.ConcreteGraphicalIU.DomainElement',
                        thisModule.getRootElement(domainElement), cm);
                }
        }
}
```

The second rule transforms a class AbstractCompoundIU into a class MenuBar.

```
rule AbstractCompoundIU2MenuBar {
        from
                aiu : AUI!AbstractCompoundIU (aiu.hasRoleApplied('menu-bar') and not
aiu.containedByMenu())
        to
                ciu : CUI!ConcreteGraphicalCompoundIU (
                    id <- aiu.id,
                    contextCondition <- aiu.role,
                    graphicalIUs <- Sequence {mBar}d rule ,
                    isNamed <- aiu.isNamed,
                    listeners <- AUI!Rule.allInstances()->select(r |
aiu.isListenBy.includes(r.trigger))
                ),
                mBar : CUI!MenuBar (
                    id <- if aiu.id <> OclUndefined and aiu.id <> '' then aiu.id +
'_menubar' else 'menubar' endif,
                    isNamed <- aiu.isNamed,
                    items <- aiu.childIU->select(iu |
                        if iu.oclIsTypeOf(AUI!AbstractTriggerIU) then
iu.eContainer().containedByMenuBarItem() else false endif or
                        if iu.oclIsTypeOf(AUI!AbstractCompoundIU) then iu.isMenuBarItem()
else false endif)
                )
        do {
                --('AbstractCompoundIU2MenuBar '+aiu.toString()+'
'+thisModule.identifier.toString()).println();

                -- change label and image id for menubar
                thisModule.changeLabelId(mBar);
                thisModule.changeImageId(mBar);
        }
}
```

The third rule transforms a class AbstractTriggerIU into CommandButton.

```
rule AbstractTriggerIU2CommandButton {
	from
		at : AUI!AbstractTriggerIU (at.isElementaryIUTransformable() and not
at.eContainer().containedByMenu())
	to
		cb : CUI!CommandButton (
			id <- at.id,
			listeners <- AUI!Rule.allInstances()->select(r |
at.isListenBy.includes(r.trigger))
		)
	do {
		--('AbstractTriggerIU2CommandButton '+at.toString()+'
'+thisModule.identifier.toString()).println();

		-- change label and image id for commandbutton
		thisModule.changeLabelId(cb);
		thisModule.changeImageId(cb);

		-- add localizations to the label contained by the commandbutton
		thisModule.addLocalizationsToTheLabel(at, cb);
	}
}
```

5.6.2. Transformation from Domain Model to AUI

We show some rules to transform Domain Model to AUI as example. The first rule transforms the class DomainModel into the class AbstractUIModel.

```
rule DomainModel2AbstractUIModel {
	from
		i	:	UML!DomainModel
	to
		o	:	AUI!AbstractUIModel (
			compoundIUs <- Sequence {o2}
		),
		o2	:	AUI!AbstractCompoundIU (
			id <- 'menu',
			role <- 'home;menu'
		)
	do {
		thisModule.model <- o;
		thisModule.menu <- o2;
	}
}
```

The second rule transforms the class Property into AbstractDataIU.

```
rule Property2AbstractDataIU {
	from
		i	:	UML!Property (
				i.eContainer().oclIsTypeOf(UML!Class)
		)
	to
		o1	:	AUI!AbstractDataIU (
			id <- thisModule.resolveTemp(i.eContainer(), 'o1').id + '_' +
i.name.toLower(),
			domainReference <- i.__xmiID__,
			parentIU <- thisModule.resolveTemp(i.eContainer(), 'o1')
		),
		o2	:	AUI!AbstractDataIU (
```

```
                    id <- thisModule.resolveTemp(i.eContainer(), 'o2').id + '_' +
i.name.toLower(),
                    domainReference <- i.__xmiID__,
                    parentIU <- thisModule.resolveTemp(i.eContainer(), 'o2')
             ),
             o3     :      AUI!AbstractDataIU (
                    id <- thisModule.resolveTemp(i.eContainer(), 'o3').id + '_' +
i.name.toLower(),
                    domainReference <- i.__xmiID__,
                    parentIU <- thisModule.resolveTemp(i.eContainer(), 'o3')
             )
      do {
             --('Property2ADIU '+i.toString()).println();

             thisModule.createAbstractLocalizations(i.name.toLower(), o1);
             thisModule.createAbstractLocalizations(i.name.toLower(), o2);
             thisModule.createAbstractLocalizations(i.name.toLower(), o3);
      }
}
```

5.6.3. Transformation from CUI to Code

We show rules to transform a ConcreteGraphycalCompoundIU into a table whose rows are composed of a list of elements

```
«IMPORT cui»

«EXTENSION m2t::xtend_files::extensions»


«DEFINE main(String appName, String basePackage) FOR cui::ConcreteGraphicalCompoundIU»
«LET getDomainElement(this).name AS entityName»
«LET getWidgetIdByActionType(this, "form-create") AS create»
«LET getWidgetIdByActionType(this, "show") AS read»
«LET getWidgetIdByActionType(this, "form-update") AS update»
«LET getWidgetIdByActionType(this, "modelDelete") AS delete»
«FILE "src/main/webapp/WEB-INF/views/" + entityName + "/list.jspx"»
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:c="http://java.sun.com/jsp/jstl/core"
       xmlns:table="urn:jsptagdir:/WEB-INF/tags/form/fields"
       xmlns:page="urn:jsptagdir:/WEB-INF/tags/form"
       xmlns:jsp="http://java.sun.com/JSP/Page"
       xmlns:spring="http://www.springframework.org/tags" version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8"/>
      <jsp:output omit-xml-declaration="yes"/>
«EXPAND div(basePackage, create, read, update, delete) FOREACH this.graphicalIUs.select(iu
| cui::ConcreteGraphicalCompoundIU.isInstance(iu))»
</div>
«ENDFILE»
«ENDLET»
«ENDLET»
«ENDLET»
«ENDLET»
«ENDLET»
«ENDDEFINE»


«DEFINE div(String basePackage, String create, String read, String update, String delete)
FOR cui::ConcreteGraphicalCompoundIU»
«IF this.contextCondition == "row"»
      «EXPAND table(basePackage, create, read, update, delete) FOR this»
«ELSE»
      <div id="«this.id»" class="widget_container">
```

```
	«EXPAND div(basePackage, create, read, update, delete) FOREACH
this.graphicalIUs.select(iu | cui::ConcreteGraphicalCompoundIU.isInstance(iu))»
	</div>
«ENDIF»
«ENDDEFINE»


«DEFINE table(String basePackage, String create, String read, String update, String delete)
FOR cui::ConcreteGraphicalCompoundIU»
«LET basePackage.replaceAll("\\.", "_") AS package»
«LET getDomainElement(this) AS entity»
	<page:list id="pl_«package»_domain_«entity.name»_l"
items="${«firstToLowerCase(entity.name)»s}">
		<table:table data="${«firstToLowerCase(entity.name)»s}"
id="l_«package»_domain_«entity.name»" path="/«entity.name»"
typeIdFieldName="internalIdentifier" create="«create»" read="«read»" update="«update»"
delete="«delete»" cssClass="id_«this.id»">

		«FOREACH this.graphicalIUs.select(w |
cui::ConcreteGraphicalElementaryIU.isInstance(w)) AS widget»
			«LET getDomainElement(widget) AS property»
			«IF property != null»
				«LET firstToLowerCase(property.name) AS propertyName»
				«IF property.type.name == "Date"»
				<table:column date="true"
dateTimePattern="${«firstToLowerCase(entity.name)»_«propertyName»_date_format}"
id="c_«package»_domain_«entity.name»_«propertyName»_l" property="«propertyName»"/>
				«ELSE»
				<table:column
id="c_«package»_domain_«entity.name»_«propertyName»_l" property="«propertyName»"/>
				«ENDIF»
				«ENDLET»
			«ELSEIF (cui::Label.isInstance(widget) && !isDomainLabel(widget)) ||
!cui::Label.isInstance(widget)»
				«EXPAND src_main_webapp_webinf_views_generic::widget FOR widget»
			«ENDIF»
			«ENDLET»
		«ENDFOREACH»

		</table:table>
	</page:list>
«ENDLET»
«ENDLET»

«ENDDEFINE»
```

6. APPLYING USABILITY CRITERIA TO CHOOSE TRANSFORMATIONS RULES

According to the UsiXML Transformation meta-model, transformation rules are chosen depending on quality criteria that we want to optimize in a specific context. According to ISO 9126-1 [ISO01], usability is a software characteristic that defines the quality of the system. In this section, we identify usability criteria that can affect the choice of a transformation rule when there are different options. These criteria have been extracted from the ISO 9126-1 [ISO01] and ergonomic criteria [BS93]. This is the list of usability criteria we have considered:

- **Information density**: The grade in which the system displays/demands the user the advisable amount of information, for each interaction unit/screen.

- **Navigability**: Guides the user by providing mechanisms such as path and current position. Absence of cyclic navigations among abstract user interfaces.
- **Brevity**: Related to patterns that reduce the cognitive effort of the user.
- **Help facilities**: Utilities supplied by the system in relation to help.
- **Default values**: Initial values to reduce the effort of the user by anticipating the most probable value for a field.
- **Grouping cohesiveness**: Visually remarking the relation among many things (fields, controls, etc.) which have something in common.

The list of usability criteria and how they affect the transformation rules is useful to define transformations considering the QOC package (see UsiXML D1.3, Transformation meta-model). This package helps to choose transformation rules depending on quality criteria we would like to optimize. Next, we relate usability criteria to transformation rules. We have focused this version of the document on the list of transformation rules defined by Delacre [Lac07] and Limbourg [LV09]. Both authors defined transformation rules based on graphs. The number that identifies each rule is the same identifier used by these authors in their proposals. We have only considered rules that can be affected by usability criteria. This is the reason why rule identifiers are not consecutive.

6.1. Task Model to Abstract User Interface Model (Task → AUI)

First, we focus on rules to transform a Task Model into an Abstract User Interface Model:

- ***Rule 1***: For each leaf task of a task tree, create an Abstract Individual Element. For each task parent of a leaf task, create an Abstract Container. Link the Abstract Container and the Abstract Individual Element by a containment relationship.
- ***Rule 2***: For each task defined in the Task Model, create an Abstract Individual Component in the AUI.
- ***Rule 3***: for each Abstract Individual Element mapped onto a task the nature of which consists in the activation of an operation and this task is mapped onto a class, assign to the abstract individual component an action facet that activates the mapped method.

Each leaf of the Task Model is transformed into an Abstract Individual Element of the Abstract User Interface Model, which can be of different types: input, output, operation, or navigation. The decision of the type depends on an attribute of the meta-class Task called canonicalTaskType (see UsiXML D1.3, Task meta-model). Apart from this attribute, the meta-class TaskDecoration with all its attributes can also guide the transformation into the Abstract Individual Elements. This class describes by means of its attributes the decoration of a task. Next, we identify how the attributes of the class TaskDecoration can affect the transformation rules 1, 2 and 3 depending on the usability criteria to optimize:

- ***Optional*** defines whether or not the task is optional. The quality criterion related to this attribute are the following:
    - **Information density**: If we would like to decrease the number of widgets in the interface, we can hide optional elements or make them accessible with navigation buttons.
    - **Navigability**: If we would like to develop a system with navigation buttons, optional tasks should be accessible by means of navigations.
    - **Brevity**: If we would like to optimize brevity, optional tasks can be displayed in the same interface than non-optional tasks. This way, the user can complete these tasks without navigations and decreasing the steps to complete all the tasks.
- ***Iterative*** defines the number of iterations that can perform the task.

- **Help facilities**: If we are modelling a system with help mechanisms to guide the user, we can hide this guide in an iterative task once it has been finished successfully in an iteration since we can assume that the user knows how to perform the task.
- **Default values**: If we would like to minimize the users' effort, we can show default values for those visual elements whose value depend on previous iterations of the task.

- ***Criticity*** defines the criticity level of the task (from 0 to 10).
  - **Grouping cohesiveness**: If we would like to optimize the groups defined by the analyst, then most critical tasks may not appear first in the interface if the analyst has not ordered the elements in the interface by criticity. However, if we consider that the groups defined by the analyst are not very important, then we can display most important tasks first in the interface, independently of the analyst's decision.
- ***Frequency*** defines the frequency of the task.
  - **Help facilities**: If we would not like to optimize the help, we can hide the description for the most frequent tasks, since these tasks will be very well-known by the user.
  - **Information density**: If we would like to optimize the information density, we can hide the description for the most frequent tasks.
- ***Centrality***: This value defines whether or not the task is important for the current context.
  - **Grouping cohesiveness**: If we would like to optimize the defined group of tasks (such as they were defined), then the most important tasks may not appear first in the interface. However, if we consider that this criterion is not important, then most important tasks will appear first in the interface (independently of how they were defined in the Task model).

For example, if we model a system to manage a rent-car system, the interface will be not the same for a PDA system or for a desktop system. Users of this system are the employees of the company. PDA system involves maximizing quality criteria related to minimize the information displayed and the actions needed to trigger a service. Moreover, we would like to distinguish between two types of users, expert users of the system and non-experts. To illustrate this, we model a form to create a new customer. Next we specify which transformation rules should be used for each platform (PDA or desktop application) and for each type of user according to quality criteria:

- For a PDA system: Since the screen in a PDA is so narrow, we would like to decrease the number of widgets in the interface. Therefore, we will hide optional tasks related to the creation of a new customer. For example, the task to register the bank account of the customer will be shown only if the end-user navigates to a specific interface. But it is not displayed in the same interface as the personal data of the customer is provided. With this approach we are worsening the quality criteria called Brevity, but this criterion is not the target for a PDA system.
- For a desktop application: The screen is not a problem for this kind of applications. Therefore, Optional tasks can be displayed in the interface, in such a way that they are easily accessible, maximizing the quality criteria called Brevity. In our illustrative example, the bank account is registered in the same interface where personal data is provided.
- For novice users: Iterative tasks should have help facilities to guide the end-users, since they are novice. In our example, the task to create a customer should have a help facility even though it will be used frequently. Moreover, default values should be provided for those widgets whose value is predictable.
- For expert users: Iterative tasks should not include help facilities, since users know how to perform the task. In our example, the task to create a customer should not have a help facility since it will be used frequently.

As conclusion, we can state that the criteria defined in the QOC model guides the rules selection. When there are different rules to be applied, we must query the defined quality criteria in order to optimize the system usability. More details about how to specify quality criteria and how these criteria are related to UsiXML transformations can be queried at D 1.3.

6.2. Abstract User Interface Model to Concrete User Interface Model (AUI → CUI)

Second, we focus on rules to transform an Abstract User Interface Model into a Concrete User Interface Model. The rules to perform this transformation that can be affected by usability criteria are the following [LV09] [Lac07]:

- ***Rule 6***: Each abstract container at level "leaf-l" is transformed into a window. Note that an abstract container is always reified into a, so called, box at the concrete level. This box is then embedded into a window.
- ***Rule 7***: Each abstract container contained into an abstract container that was reified into a window is transformed into a horizontal box and embedded into the window.

The abstract containers that are transformed into a horizontal box can be displayed in different order. This order is related to the usability criterion **Grouping cohesiveness**. Depending on the value we would like to optimize, we can order the elements by format or by location (such as these elements were defined in the abstract container).

- ***Rule 8***: Each input facet of an abstract individual component is reified by a graphical individual component (a type of concrete individual component) of type "editable text component" (i.e., a text box).

According to Rule 8, abstract input elements defined in the Abstract User Interface Model must be transformed into concrete graphical elements in the Concrete User Interface Model. Next, we identify quality criteria that can affect the selection of the target concrete graphical element:

  - **Information density**: If we would like to show as few elements per interface as possible, input elements should be transformed into a list. The components of the list will be displayed only when the user opens the list.
  - **Brevity**: If we would like to minimize the actions that the user must perform in the system, lists are not a good choice, since the users must open these lists to display all their elements. In this case, for input elements that only can accept a restricted number of elements, a RadioBox is more suitable than a list.

- ***Rule 10***: For each container related to another container belonging to different windows and their respective abstract container related by a "is before relationship", generate a navigation button in source container pointing to the window of target container.

Depending on the usability criteria we would like to optimize, the target container can be included on the source container. This way, source and target containers will be displayed in the same interface. Next, we detail the quality criteria to decide whether or not a navigation is the best option.

  - **Information density**: If we would like to optimize this criterion, the navigation relationship between both containers is the best option. The information of the target container will be only displayed when the user performs the navigation.
  - **Navigability**: If we would like to build a navigable system, the navigation relationship between both containers is also the best option.
  - **Brevity**: If we would like to minimize the users' effort to interact with the system, we must provide as much information as possible in the same interface. Therefore, source and target containers could be included in the same interface. Each container can be displayed in a group box or in a tab, but in the same interface.

As an illustrative example, we are going to use the same scenario used in the previous section, a system to manage a rent-car system, and more specifically, the interface to create a customer. As previously mentionned, applications for PDA should optimize the information displayed in the interface, but this restriction does not appear in desktop applications. Quality criteria to optimize the application for a PDA or for a desktop system will guide the transformation rules used in the transformation from AUI to CUI:

- For a PDA system: The quality criterion called Information density should be maximized. In our example, the input element that represents the marital status in the AUI will be transformed into a SelectBox in the CUI. This way, the list of possible values for the input element called Marital Status is displayed in the interface only when the end-user clicks on the SelectBox. In order to minimize the displayed elements in the interface, the system should implement navigations to more information. For example, the form to insert a bank account can be reachable by means of a navigation from the interface to create a customer.
- For a desktop application: Brevity criterion is more important than Information density. Therefore, in our example, the input element that represents the marital status in the AUI will be transformed into a RadioButton in the CUI. The items of the RadioButton are visible without performing any action (the SelectBox needs a click on the list to display its elements), therefore this widget improves Brevity but decreases Information Density. In desktop applications, we will need less navigation, since the widgets to insert a bank account can be reachable in the interface to create a customer.

7. CONCLUSIONS

This document defines some transformations throughout UsiXML models. These transformations are grouped by UsiXML models. There is more than one transformation between two models, and some of these transformations can be contradictory to each other. In order to select the most optimum transformation rule when there is more than one option, we have identified some quality criteria. These quality criteria aim to guide the analyst in the decision to choose transformation rules. Moreover, we have identified different paths to develop a system from the most abstract model of UsiXML until code. Depending on the end-user's needs, the analyst must choose the most suitable path. As future work, we plan to specify more transformation rules and to identify more possible paths among all the models that compose the UsiXML framework.

REFERENCES

[BS93]

J. M. Bastien, Scapin, D., "Ergonomic Criteria for the Evaluation of Human-Computer Interfaces," Rapport technique de l'INRIA, pp. 79, 1993.

[Cer11]

Céret E., 2011, *Toward a Flexible Design Method Sustaining UIs Plasticity*, In Doctoral consortium of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS'2011), Pisa, Italy, June, 13th-16th.

[GCD+11]

García Frey, A., Céret E., Dupuy-Chessa, S., Calvary, G., 2011, *QUIMERA: A Quality Metamodel to Improve Design Rationale*, In Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS'2011), Pisa, Italy, June, 13th-16th.

[ISO01]

ISO/IEC 9126-1 (2001), Software engineering - Product quality - 1: Quality model.

[Lac07]

J.-P. DeLacre, "A Comparative Analysis of Transformation Engines for User Interface Development." Lovain: Université catholique de Louvain, 2007.

[LV09]

Q. Limbourg and J. Vanderdonckt, "Multipath Transformational Development of User Interfaces with Graph Transformations," in Human-Centered Software Engineering, vol. II: Springer, 2009, pp. 107-138.

[Nor91]

Norman,D.A. (1991) “Cognitive artifacts”. J.M.Carroll (ed.) Designing Interaction - Psychology at the Human-Computer Interface. Cambridge University Press.

[Pat99]

Paternò, F.(1999) Model-based design and evaluation of interactive applications, Springer-Verlag, Berlin.

[Pri06a]

Pribeanu, C. (2007) “Tool support for handling mapping rules from domain to task models”. Coninx, K., Luyten, K. & Schneider, K. (Eds.): Proceedings of TAMODIA 2006, Hasselt, Belgium, 23-24 October 2006. Lecture Notes in Computer Science - LNCS 4385, Springer 2007, 16-23.

[Pri06b]

Pribeanu, C. (2006) Task Modeling for User Interface Design – A Layered Approach, International Journal of Information Technology, 3(2), 86-90.

[Pri09]

Pribeanu, C. (2009) “A usability assistant for the heuristic evaluation of interactive systems”. Studies in Informatics and Control, 18(4), 355-362

[Ralyté 2001]

Jolita Ralyté and Colette Rolland. An assembly process model for method engineering. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, CAiSE, volume 2068 of Lecture Notes in Computer Science, pages 267–283. Springer, 2001.

[Brinkkemper 1998]

Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Assembly techniques for method engineering. In Barbara Pernici and Costantino Thanos, editors, CAiSE, volume 1413 of Lecture Notes in Computer Science, pages 381–400. Springer, 1998.

[Crescenzo 2009]

Isabelle Mirbel and Pierre Crescenzo. Improving collaborations in neuroscientist community. In Web Intelligence/IAT Workshops, pages 567–570. IEEE, 2009.

[Perez-Medina 2010]

Jorge Luis Pérez-Medina, Sophie Dupuy-Chessa, and Dominique Rieu. A service-oriented approach for interactive system design. In David England, Philippe A. Palanque, Jean Vanderdonckt, and Peter J. Wild, editors, TAMODIA, volume 5963 of Lecture Notes in Computer Science, pages 44–57. Springer, 2009.

[Garzotto 2007]

Garzotto, F., Perrone, V.: Industrial acceptability of web design methods: an empirical study. J. Web Eng. 6 (2007) 73-96

[Fitzgerald 1998]

Fitzgerald, B.: An empirical investigation into the adoption of systems development methodologies. Inf. Manage. 34 (1998) 317-328.

[Barry 2001]

Barry, C., Lang, M.: A Survey of Multimedia and Web Development Techniques and Methodology Usage. IEEE MultiMedia 8 (2001) 52-60.

[Rolland 2006]

Rolland, C.: From Conceptual Modeling to Requirements Engineering. Conceptual Modeling - ER 2006 (2006) 5—11

[ATL]

http://www.eclipse.org/atl/